This repository Search or type a command Explore Gist Blog Help corbett

WhisperSystems / TextSecure-Server Watch 6 Star 33 Fork 5

Home Pages History New Page

Edit Page Page History Clone URL

API Protocol

Registration

Request a verification code

```
GET /v1/accounts/{transport}/code/{number}
```

The client requests an SMS or Voice verification code for the client's PSTN number.

1. `transport` is the string `sms` or `voice`, depending on how the client would like a verification code delivered.
2. `number` is the client's PSTN number.

Returns:

1. `200` request was processed successfully.
2. `400` badly formatted `number`.
3. `415` invalid `transport`.
4. `413` rate limit exceeded. Too many requests.

Confirm a verification code

```
PUT /v1/acccounts/code/{verification_code}
Authorization: Basic {basic_auth}

{
  "signalingKey" : "{bas64_encoded_52_byte_key}"
  "supportsSms" : false
}
```

The client submits the verification code it received via voice or SMS to the server for confirmation.

1. `verification_code` is the code it received via voice or SMS, numeric only.
2. `basic_auth` are the authorization credentials the client would like to create. These are in the form of `Base64({number}:{password})`, where `number` is the client's verified PSTN number and `password` is a randomly generated 16 byte string.
3. `signalingKey` is a randomly generated 32 byte AES key and a 20 byte HMAC-SHA1 MAC key, concatenated together and Base64 encoded.
4. `supportsSms` indicates whether a client supports SMS as a transport.

Returns:

1. `200` account successfully verified.
2. `401` badly formatted `basic_auth`.
3. `403` incorrect `verification_code`.
4. `413` rate limit exceeded.

Registering an APN or GCM id

```
PUT /v1/accounts/apn/
Authorization: Basic {basic_auth}

{
  apnRegistrationId: "{apn_registration_id}"
}
```

or

```
PUT /v1/accounts/gcm/
Authorization: Basic {basic_auth}

{
  gcmRegistrationId: "{gcm_registration_id}"
}
```

The client submits its APN or GCM push registration ID.

1. `basic_auth` is the client's authorization credentials (see above).
2. `gcm_reistration_id` or `apn_registration_id` is the client's registration ID.

Returns:

1. `200` request succeeded.
2. `401` invalid authentication credentials.
3. `415` badly formatted JSON.

Registering prekeys

```
PUT /v1/keys/
Authorization: Basic {basic_auth}

{
  lastResortKey : {
                   keyId: 0xFFFFFF
                   publicKey: "{public_key}"
                   identityKey: "{identity_key}"
                  },
   keys: [
         {
          keyId: {key_id},
          publicKey: "{public_key}",
          identityKey: "{public_key}"
         },
         ...]
}
```

1. `public_key` is a randomly generated Curve25519 public key with a leading byte of `0x05` to indicate its type. This is a total of 33 bytes, base64 encoded without padding (no ==).
2. `identity_key` is a Curve25519 public key with a leading byte of `0x05` to indicate its type. This is a total of 33 bytes, base64 encoded without padding (no ==). Each client should have a single identity key generated at install time.
3. `key_id` each prekey has a unique 24bit identifier. The last resort key is always 0xFFFFFF.

Returns:

1. `200` request succeeded.
2. `401` invalid authentication credentials.
3. `415` badly formatted JSON.

Getting a contact intersection

```
PUT /v1/directory/tokens
Authorization: Basic {basic_auth}
{
  "contacts": [{"{token}", "{token}", ..., "{token}"]
}
```

1. `token` is Base64(SHA1(E164number)[0:10]) without Base64 padding.

Returns:

1. `400` badly formatted token(s).
2. `401` invalid authentication credentials.
3. `415` badly formatted JSON.
4. `200` request succeeded. The structure below is returned.

```
{
  contacts: [{token="{token}", relay="{relay}", supportsSms="true"},
             {token="{token}"},
             ...,
             {token="tokenN", relay="{relay}"}]
}
```

1. `token` is Base64(SHA1(E164number)[0:10]) without Base64 padding.
2. `relay` is the name of a federated node which this contact is associated with.
3. `supportsSms` indicates that the contact supports the SMS transport.

At this point the client should be fully registered.

Sending Messages

Message Format

Messages bodies sent and received by clients are a protocol buffer structure:

```
message PushMessageContent {
  optional string body = 1;

  message AttachmentPointer {
    optional fixed64 id = 1;
    optional string contentType = 2;
    optional bytes key = 3;
  }

  repeated AttachmentPointer attachments = 2;
}
```

Getting a recipient's PreKey

If a client does not have an existing session with a recipient, the client will need to retrieve a PreKey for the recipient in order to start one.

```
GET /v1/keys/{number}?relay={relay}
Authorization: Basic {basic_auth}
```

1. `number` is the PSTN number of the recipient.

2. `relay` (optional) is the federated relay the recipient is associated with. The `relay` param should only be included if the destination is at a federated node other than the sender.

Returns:

1. `401` invalid authentication credentials.
2. `413` rate limit exceeded.
3. `404` unknown/unregistered `number`.
4. `200` request succeeded. The structure below is returned.

```
{
  keyId: {key_id},
  publicKey: "{public_key}",
  identityKey: "{public_key}"
}
```

Submitting a message

```
POST /v1/messages/
Authorization Basic {basic_auth}

{
  messages: [{
               type: {type},
               destination: "{destination_number}",
               body: "{base64_encoded_message_body}", // Encrypted PushMessageContent
               relay: "{relay}",
               timestamp: "{time_sent_millis_since_epoc}"
             },
            ...,
            ]
}
```

1. `type` is the type of message. Supported types are enumerated below.
2. `destination_number` is the PSTN number of the message recipient.
3. `body` is the Base64 encoded (without padding) and encrypted `PushMessageContent` (above).
4. `relay` (optional) is the relay the message recipient is registered with.
5. `timestamp_sent_millis_since_epoch` is the timestamp of the message in millis since the epoch.

Returns:

1. `401` invalid authentication credentials.
2. `413` rate limit exceeded.
3. `415` badly formatted JSON.
4. `200` request succeeded. The structure below is returned.

```
{
  "success" : [{destination_number}, {destination_number}, ..., {destination_number}],
  "failure" : [{destination_number},...,{destination_number}]
}
```

Supported types:

```
int TYPE_MESSAGE_PLAINTEXT     = 0;
int TYPE_MESSAGE_CIPHERTEXT    = 1;
int TYPE_MESSAGE_PREKEY_BUNDLE = 3;
```

Receiving a message

APN clients will receive a push notification:

```
{
  alert: "You have a new message!",
  "m": "{payload}"
}
```

GCM clients will receive a push notification:

```
{payload}
```

1. `payload` is a Base64 encoded (without padding) `IncomingPushMessageSignal`, which is encrypted and MAC'd using the `signalingKey` submitted during registration.

Encrypted IncomingPushMessageSignal format:

```
struct {
  opaque version[1];
  opaque iv[16];
  opaque ciphertext[...]; // The IncomingPushMessageSignal
  opaque mac[10];
```

The IncomingPushMessageSignal protocol buffer:

```
message OutgoingMessageSignal {
  optional uint32 type         = 1;
  optional string source       = 2;
  optional string relay        = 3;
  repeated string destinations = 4;
  optional uint64 timestamp    = 5;
  optional bytes message       = 6; // Encrypted PushMessageContent (above)
}
```

Attachments

Recall that a push message is transmitted as the following structure:

```
message PushMessageContent {
  optional string body = 1;

  message AttachmentPointer {
    optional fixed64 id = 1;
    optional string contentType = 2;
    optional bytes key = 3;
  }

  repeated AttachmentPointer attachments = 2;
}
```

To fill out the `AttachmentPointer` structure, the client takes the following steps:

1. Generates a single-use 32 byte AES key and 32 byte Hmac-SHA256 key.
2. Encrypts the attachment using AES in CBC mode with PKCS#5 padding and a random IV, then formats the encrypted blob as `IV || Ciphertext || MAC`.
3. Requests an attachment allocation from the server.
4. Uploads the attachment to the allocation.
5. Constructs the `AttachmentPointer` with the attachment allocation `id`, the attachment's MIME `contentType`, and the concatenated 32 byte AES and 32 byte Hmac-SHA256 `key`.

Allocating an attachment

```
GET /v1/attachments/
Authorization: {basic_auth}
```

Returns:

1. `401` invalid authentication credentials.
2. `413` rate limit exceeded.
3. `200` request succeeded. The structure below is returned.

```
{
  "id" : "{attachment_id}",
  "location" : "{attachment_url}"
}
```

Uploading an attachment

```
PUT {attachment_url}
Content-Type: application/octet-stream
```

The client `PUT`s the encrypted binary blob to the `attachment_url` returned from the attachment allocation step.

Retrieving an attachment

```
GET /v1/attachments/{attachment_id}
Authorization: {basic_auth}
```

1. `attachment_id` is the `id` in a received `AttachmentPointer` protocol buffer.

Returns

1. `401` invalid authentication credentials.
2. `413` rate limit exceeded.
3. `200` request succeeded. The structure below is returned.

```
{
  "location" : "{attachment_url}"
}
```

The client can now `GET {attachment_url}` to retrieve the encrypted binary blob.

Last edited by Moxie Marlinspike, 14 days ago

© 2013 GitHub, Inc. Terms Privacy Security Contact

Status API Training Shop Blog About